Virus Bulletin :: Compromised library

By Raul AlvarezFortinet, CanadaEditor: Helen Martin

Archived: 2026-04-05 22:23:11 UTC

2012-12-03

Abstract

The Floxif DLL file infector implements both anti-static- and anti-dynamic-analysis techniques. Raul Alvarez describes how.

Table of contents

In the October issue of *Virus Bulletin* [1] I wrote about the Quervar file infector, which infects .EXE, .DOC, .DOCX, .XLS and .XLSX files. We have seen hundreds of file infectors that can infect executable files, and we also have seen document-infecting malware. However, Quevar infects document files not because they are documents, but because they have the extension used by document files – if you rename any file with '.DOC' or '.XLS' as the first three letters of the extension name, chances are, they would be infected.

Just a few weeks after Quervar, we discovered a file infector whose main target is DLL files. The malware code is not highly encrypted, but it has some interesting sophistication. This article focuses on the DLL file infector dubbed Floxif/Pioneer. We will uncover how it implements both anti-static- and anti-dynamic-analysis techniques.

Executing an infected DLL

Once an infected DLL is loaded into memory, a jump instruction at the entry point of the file will lead to the malware body. This instruction is a five-byte piece of code that is added by Floxif every time it infects a DLL. The original five bytes of the host file are stored somewhere in the virus body.

Floxif starts by getting the imagebase of kernel32.dll by parsing the Process Environment Block (PEB). Once the imagebase is established, it starts parsing the exported API names of kernel32.dll, searching for 'GetProcAddress' and eventually getting the equivalent address for this API.

Once the GetProcAddress API has been found, it starts getting the API addresses of GetProcessHeap, GetModuleFileNameA, GetSystemDirectoryA, GetTempPathA, CloseHandle, CreateFileA, GetFileSize, ReadFile, VirtualProtect, LoadLibraryA and WriteFile.

Every time an API (from the list mentioned above) is needed, the virus gets its equivalent address and executes it. The following is a summary of the execution:

Floxif reserves a memory space, opens the original DLL file and loads it in a newly created space. It starts decrypting part of the virus code from the newly loaded DLL file in memory, revealing the contents of the UPX version of symsrv.dll, which will be dropped later. (Symsrv.dll plays an important role in the overall infection process.) The decryptor is a simple combination of XOR 0x2A and NOT instructions.

After decrypting the content of the symsrv.dll file, it also decrypts the strings ('C:\Program Files\Common Files\System\symsrv.dll') where the file will be dropped. After dropping symsrv.dll, Floxif will load it as one of the modules of the infected DLL file in memory using the LoadLibraryA API. (It is interesting to note that the content of symsrv.dll is already accessible by Floxif, but it still reloads symsrv.dll as a module.)

Acting as a module, Floxif can use the exported functions of symsrv.dll as some sort of API. Two exported APIs are contained in symsrv.dll, namely: FloodFix and crc32. The virus gets its name from the FloodFix API. (The crc32 API is a continuous loop to a call to a sleep function with a one-minute interval.)

Once the symsrv.dll module is properly loaded into the host DLL, the virus will execute the FloodFix API. Let's take a closer look at what this API does.

First, it changes the protection of the memory used by the host DLL between the start of the PE header and before the section header, to PAGE_EXECUTE_READWRITE. Then, it restores the virtual address and the size of the base relocation table. Afterwards, it resets the protection of the same memory area to PAGE_READONLY.

Next, it changes the protection of the whole .text section to PAGE_EXECUTE_READWRITE and restores 3,513 bytes of code. Then, it resets the protection to PAGE_EXECUTE_READ. Afterwards, it restores the original five-byte code to the host DLL entry point.

Finally, jumping to the entry point of the host DLL file, it executes the original file.

The main function of the FloodFix API is to restore the host DLL in its original form in memory and to execute the host DLL, starting at its entry point, while the virus runs in the background.

Anti-static-analysis trick

Before we go any further, let's look into Floxif's anti-static-analysis trick. If the malware code is not encrypted, or binary dumped from the decrypted code, we can quickly take a look at its functionality using static analysis. In the case of Floxif, it looks as if the code is corrupted, because a disassembler can't render it properly. Figure 1 shows what the virus code looks like if we are just browsing it.

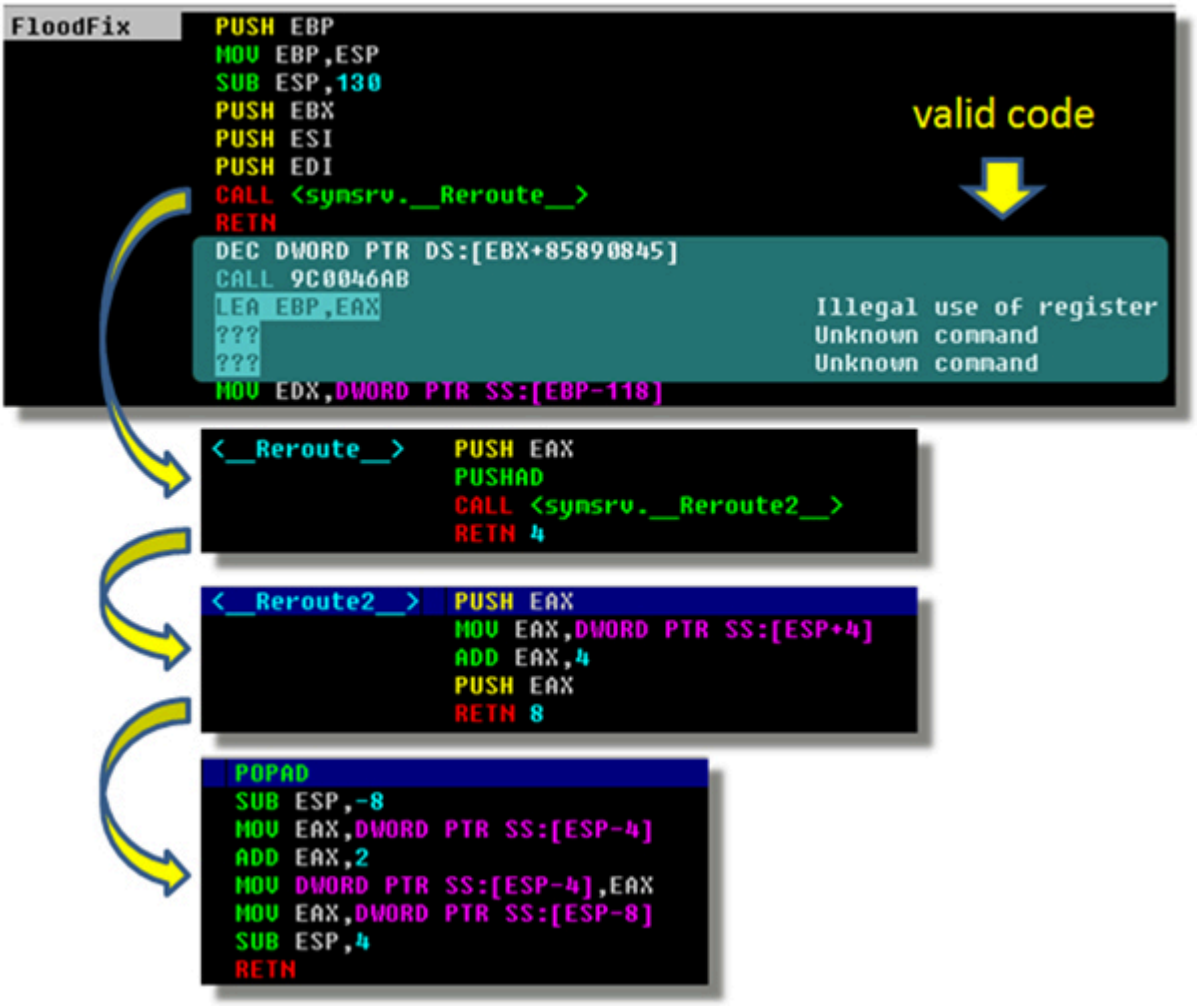

Figure 1. Browsing the virus code.

The lines of code highlighted in the figure are not junk code or corrupted data. The disassembler/debugger can't disassemble the code properly because an 'EXTRA' byte has been added after the RETN instruction. By default, the disassembler will re-interpret the code after the RETN as a new function, and it will look like junk/corrupted code.

The call to the Reroute function leads to another call, this time to the Reroute2 function. Using static analysis, a disassembler won't be able to follow the RETN 8 instruction. We can assume that it will jump back to the caller, hence we will just end up at the first call.

Using a debugger, following the RETN 8 instruction from the Reroute2 function will lead to another routine, which in turn will jump to another location – but instead of jumping to the location straight after the RETN, the new location is just after the extra byte.

Figure 2 shows the disassembler's attempt to interpret the code after the RETN following the first CALL instruction, and the equivalent code once the proper jump has been established.

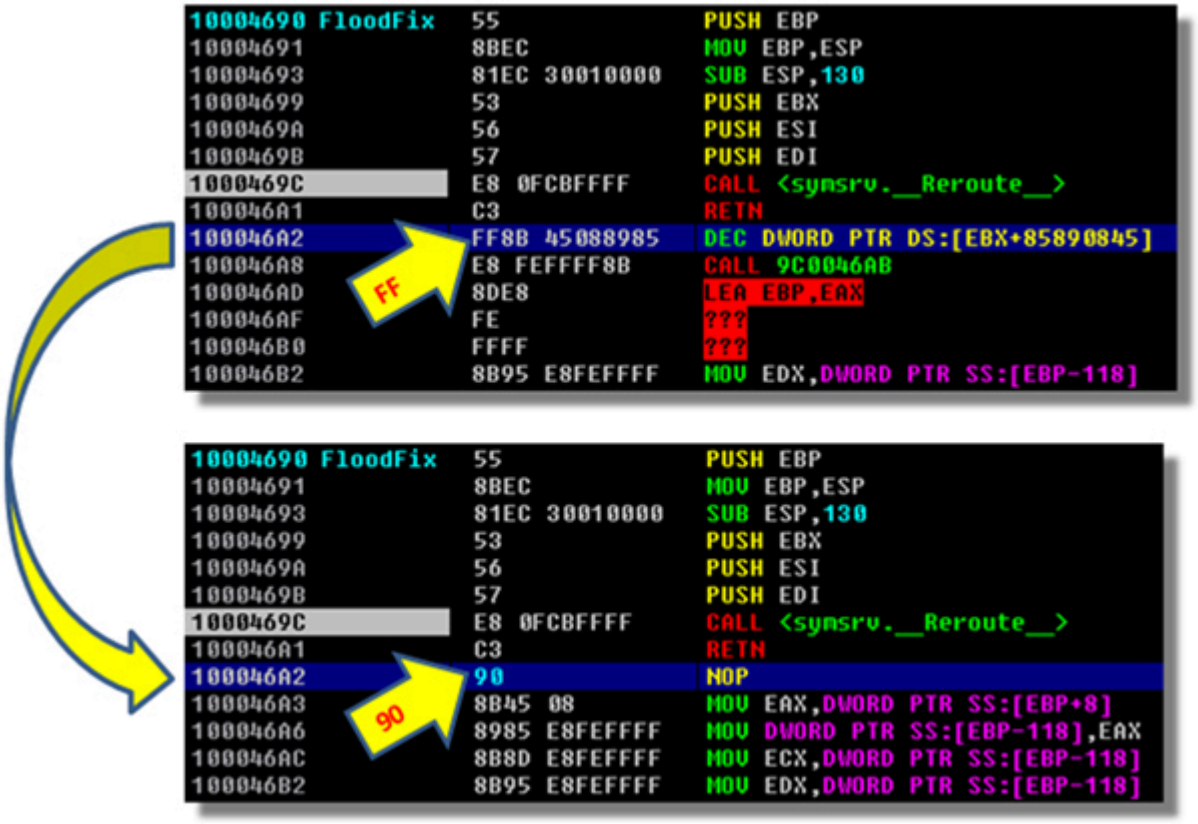

Figure 2. Disassembler's attempt to interpret code after the RETN, and equivalent code once the proper jump has been established.

The byte (FF) at address 100046A2 was added to disorient the disassembler. To emphasize the point, modifying the byte FF to 90 (NOP instruction) will yield the proper representation of the code which the CALL <symsrv.__Reroute__> will be jumping into.

This anti-static-analysis trick is an attempt to force the analyst to perform dynamic analysis using a debugger.

Anti-dynamic-analysis trick

Once we have decided that dynamic analysis is the better alternative, Floxif has another surprise.

The FloodFix API found at symsrv.dll doesn't do anything other than restoring the host DLL and its entry point. Some dynamic analysis approaches involve modifying the instruction pointer (EIP) to start at some interesting part of the code, assuming that the data and code are properly configured.

Floxif is aware of this method. To implement an anti-dynamic-analysis trick, Floxif hooks the KiUserExceptionDispatcher API of ntdll.dll. Any attempt to change the EIP to anywhere within symsrv.dll might result in the error message shown in Figure 3. Also shown in Figure 3 is the hook calling the address 10001220, which contains the function that displays an error message. After displaying the message box, the virus will terminate its execution.

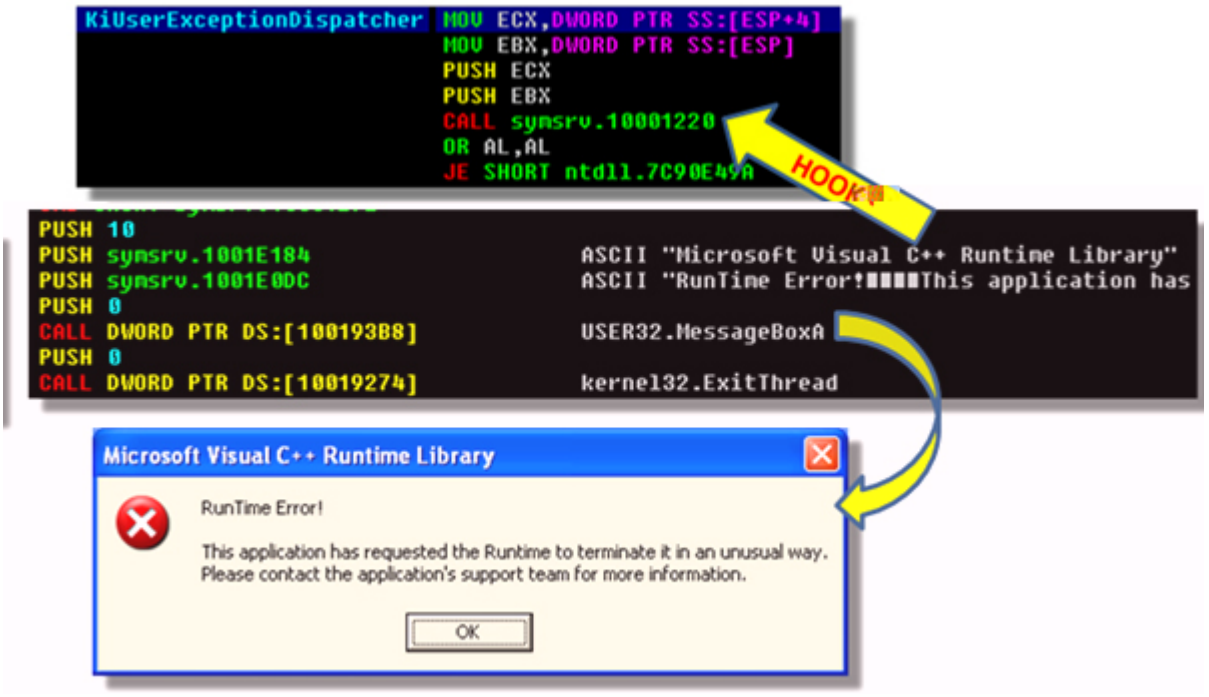

Figure 3. Hook calling the address 10001220, which contains the function that displays an error message.

This anti-dynamic-analysis trick is easy to overlook because the error message resembles a valid error message from the operating system.

Now, the infection routine

We know that the infection routine is not triggered in FloodFix or in the crc32 API. The infection routine is triggered once symsrv.dll is loaded into the memory space of the infected DLL file, using a call to the LoadLibrary API.

Thereby, the virus is already infecting the system in the background while the FloodFix API is being called.

Let’s take a look at what happens behind the scenes:

Floxif adjusts the privilege of the access token to enable it to hook the KiUserExceptionDispatcher API from ntdll.dll. The KiUserExceptionDispatcher API is used for some sort of anti-dynamic-analysis, as discussed earlier. To hook the API, it gets its virtual address by loading ntdll.dll using LoadLibraryA, then using GetProcAddress to get the API’s address.

Once the address of the KiUserExceptionDispatcher API has been acquired, the virus parses the API code looking for a jump instruction. Once found, it saves the original jump location and overwrites it with a relative value that will enable it to jump to 10001220 (Figure 3 shows the hooked location).

After hooking the KiUserExceptionDispatcher API, the virus creates a mutex named ‘Global\SYS_E0A9138’ (see Figure 4), which is initially encrypted using a NOT instruction.

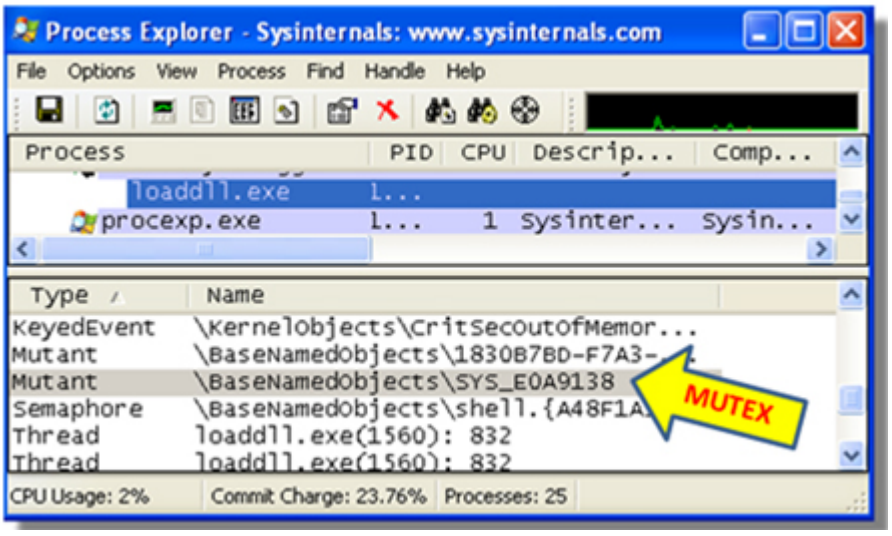

Figure 4. The virus creates a mutex.

After creating the mutex, it stores the names of the %system%, %windows% and %temp% folders using the GetSystemDirectoryA, GetWindowsDirectoryA and GetTempPathA APIs, respectively. Floxif avoids infecting files found in these folders.

Next, it starts enumerating the modules for each process running in the system. Floxif does this by getting the process list using a combination of the CreateToolhelp32Snapshot, Process32First and Process32Next APIs. It gets the module list from each process by using a combination of the CreateToolhelp32Snapshot, Module32First and Module32Next APIs.

Each module's path is checked against the three folders whose names were stored earlier: %system%, %windows%, and %temp%. Provided the module is not located in any of the three folders mentioned, the virus will read the file to memory and infect it. Then, it renames the original DLL file from <filename.DLL> to '<filename.DLL>.DAT'. Floxif then creates a new file with the infected version, which it names <filename.DLL> (i.e. the same as the original).

It will delete <filename.DLL.DAT> the next time the system is restarted by using the MoveFileExA API with the parameter NewName=NULL Flags=DELAY_UNTIL_REBOOT.

Then, the conclusion

Anti-static- and anti-dynamic-analysis techniques are not new. We encounter them on a regular, if not daily basis. There are even more sophisticated techniques than these, but we seldom see them being discussed. It is interesting to see a piece of malware that infects DLL files employing anti-analysis techniques. It is possible that I have missed other techniques that are deployed by the malware, such as anti-debugging, anti-emulation, or anti-anything-else.

What seems certain is that we are likely to see more of both Quervar and Floxif messing our files around.

Bibliography